PGdP

Nebenläufigkeit II

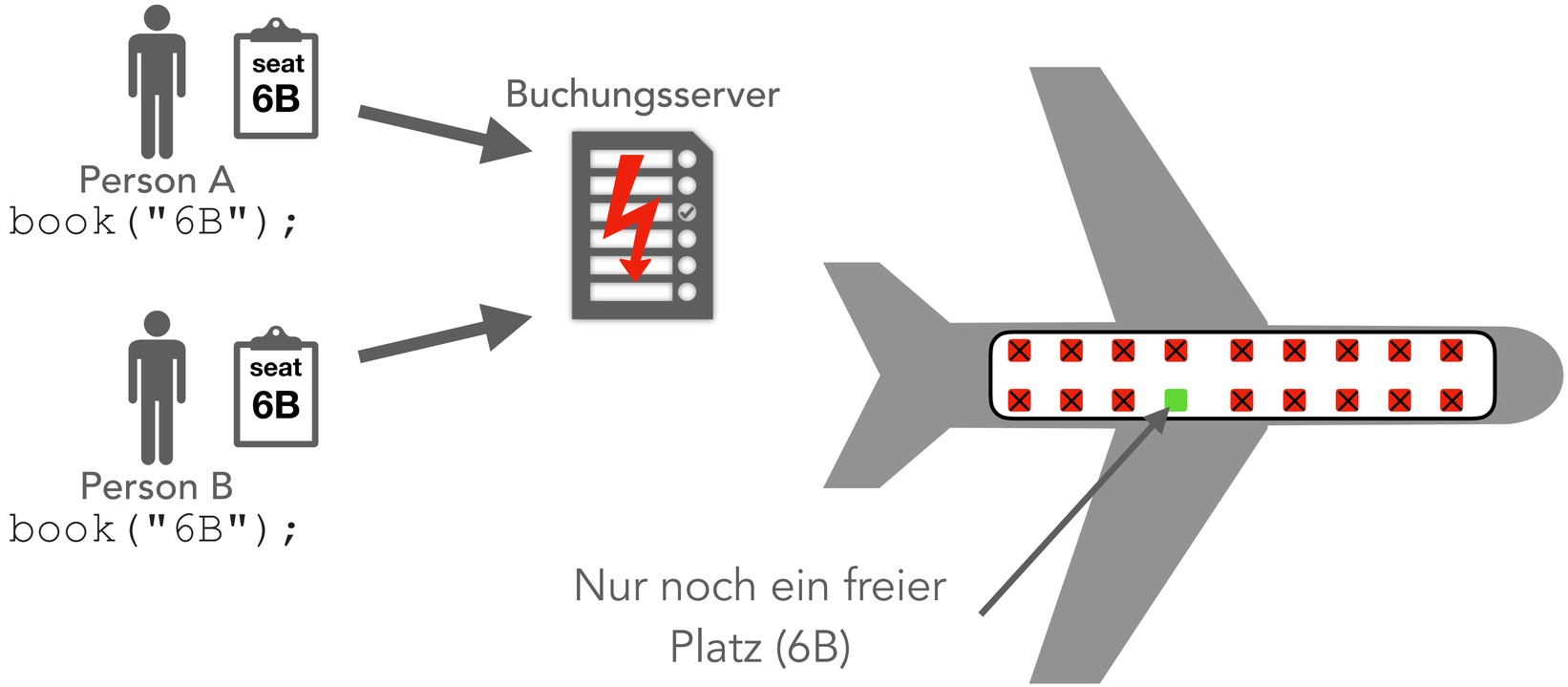
P-Aufgaben

Folien: go.tum.de/904005

Tobias Schamel

Nebenläufigkeit II

Bei nicht unabhängigen Prozessen kann es zu *Race-Conditions* kommen, Beispiel Flugbuchung



Nebenläufigkeit II

Bei nicht unabhängigen Prozessen kann es zu *Race-Conditions* kommen.

Strategien zur Verhinderung von Race Conditions

- Schlüsselwort **`synchronized`**
- Read/Write-Lock

Nebenläufigkeit II

Bei nicht unabhängigen Prozessen kann es zu *Race-Conditions* kommen: **`synchronized`**

Auf synchronized Blöcke kann immer nur ein Thread zugreifen

```
synchronized (obj) {
  // obj ist in diesem Block geschützt
}
```

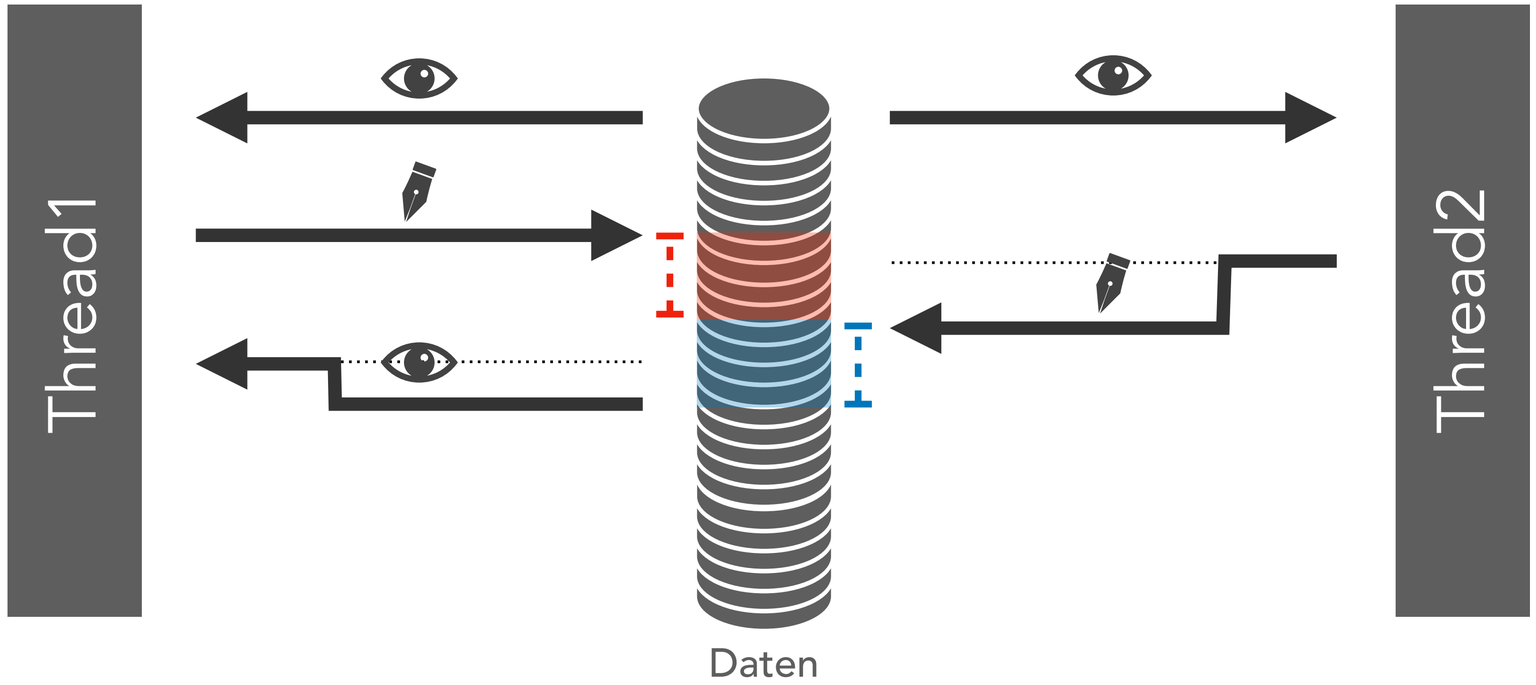
```
public static synchronized void foo();
public synchronized void foo();
// Objekt/Klasse ist geschützt
```

äquivalent zu

```
public void foo() {
 synchronized(this) {
 }
}
```

Nebenläufigkeit II

Bei nicht unabhängigen Prozessen kann es zu *Race-Conditions* kommen: ***Read/Write-Lock***



Nebenläufigkeit II

Bei nicht unabhängigen Prozessen kann es zu *Race-Conditions* kommen: ***Read/Write-Lock***

- Beliebig viele gleichzeitige Lesende Zugriffe
- Maximal ein schreibender Zugriff
  > während des Schreibens darf nicht gelesen werden

Lesen	Schreiben	Erlaubt?	Begründung
15	0	✓	Nur Lesende Zugriffe
10	1	✗	Lesende & Schreibende Zugriffe
0	2	✗	Zwei Schreibende Zugriffe
0	1	✓	Nur ein Schreibender Zugriff

Nebenläufigkeit II

Bei nicht unabhängigen Prozessen kann
Race-Conditions kommen: ***Read/Write-***

$cR = 0 \rightarrow$ kein Zugriff
$cR > 0 \rightarrow$ Anzahl Leser
$cR < 0 \rightarrow$ Anzahl Schreiber

```
public class RW {
  private int countReaders = 0;
  public synchronized void startRead() {
    while(countReader < 0) wait();
    countReaders++;
  }
  public synchronized void endRead() {
    countReaders--;
    if(countReaders == 0) notifyAll();
  }
```

Nebenläufigkeit II

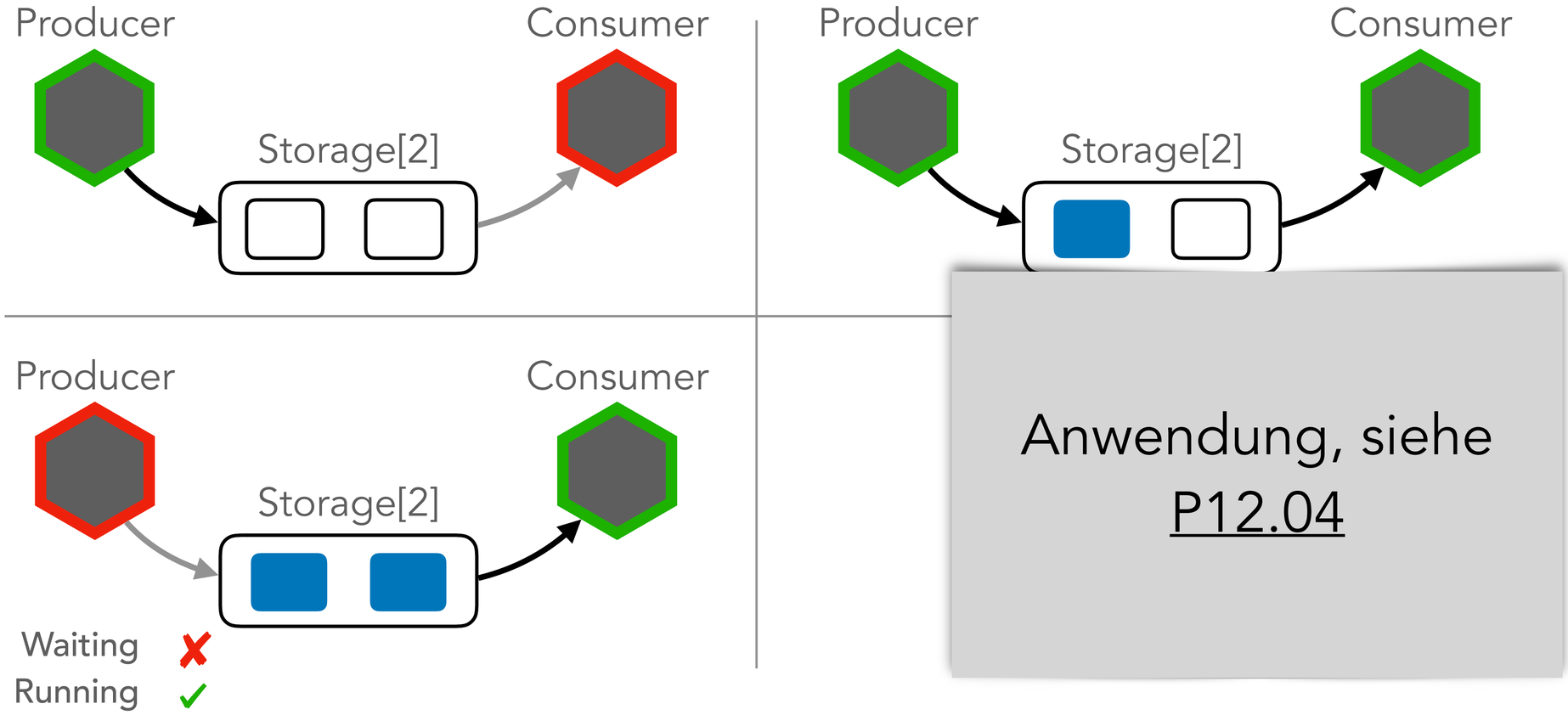
Bei nicht unabhängigen Prozessen kann
Race-Conditions kommen: ***Read/Write-***

$cR = 0 \rightarrow$ kein Zugriff
$cR > 0 \rightarrow$ Anzahl Leser
$cR < 0 \rightarrow$ Anzahl Schreiber

```
    public synchronized void
     startWrite() {
      while(countReaders != 0) wait();
      countReaders = -1;
    }
    public synchronized void endWrite() {
      countReaders = 0;
      notifyAll();
    }
}
```

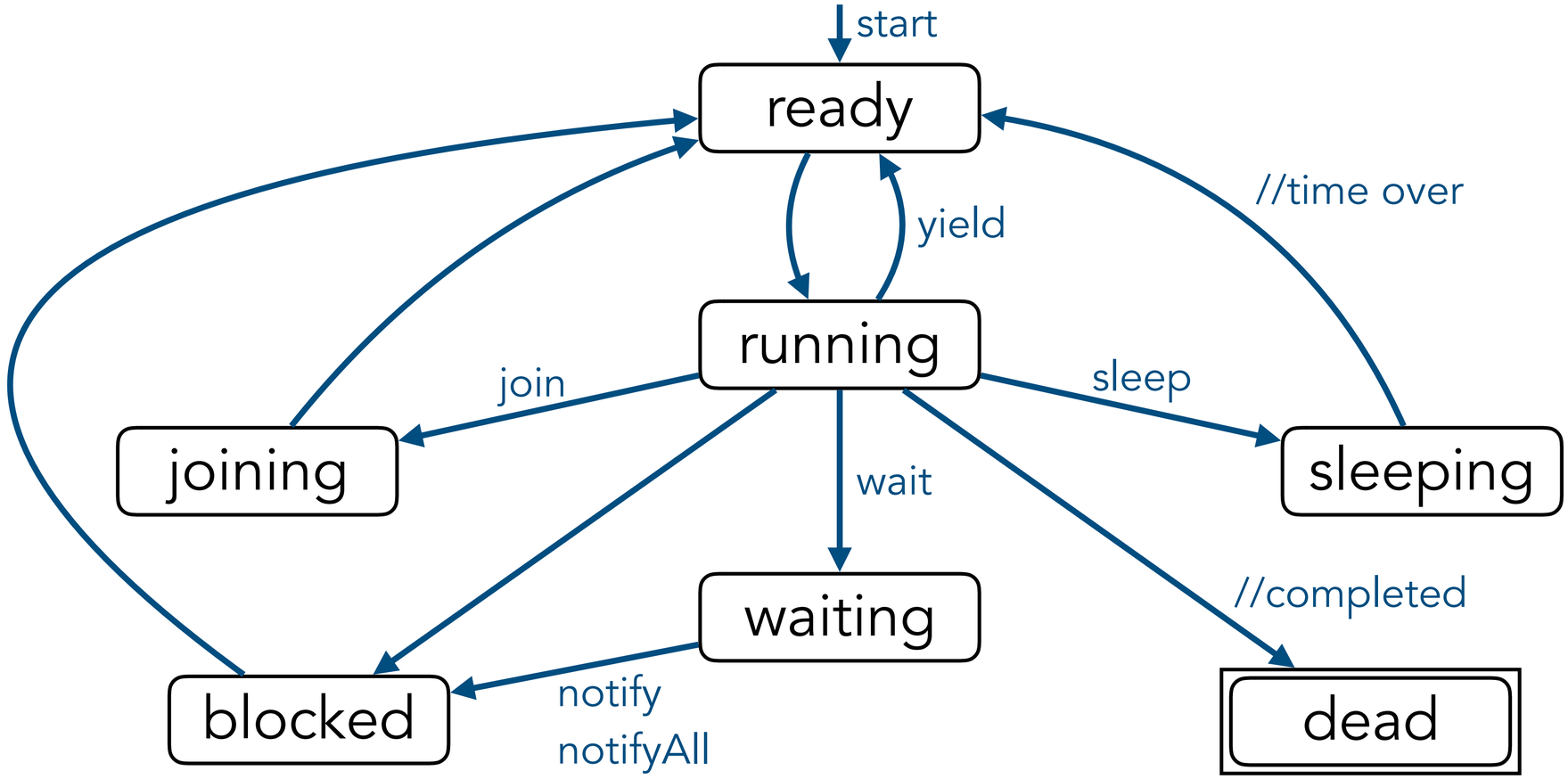
Nebenläufigkeit II

Bei nicht unabhängigen Prozessen kann es zu *Race-Conditions* kommen: ***Producer/Consumer***



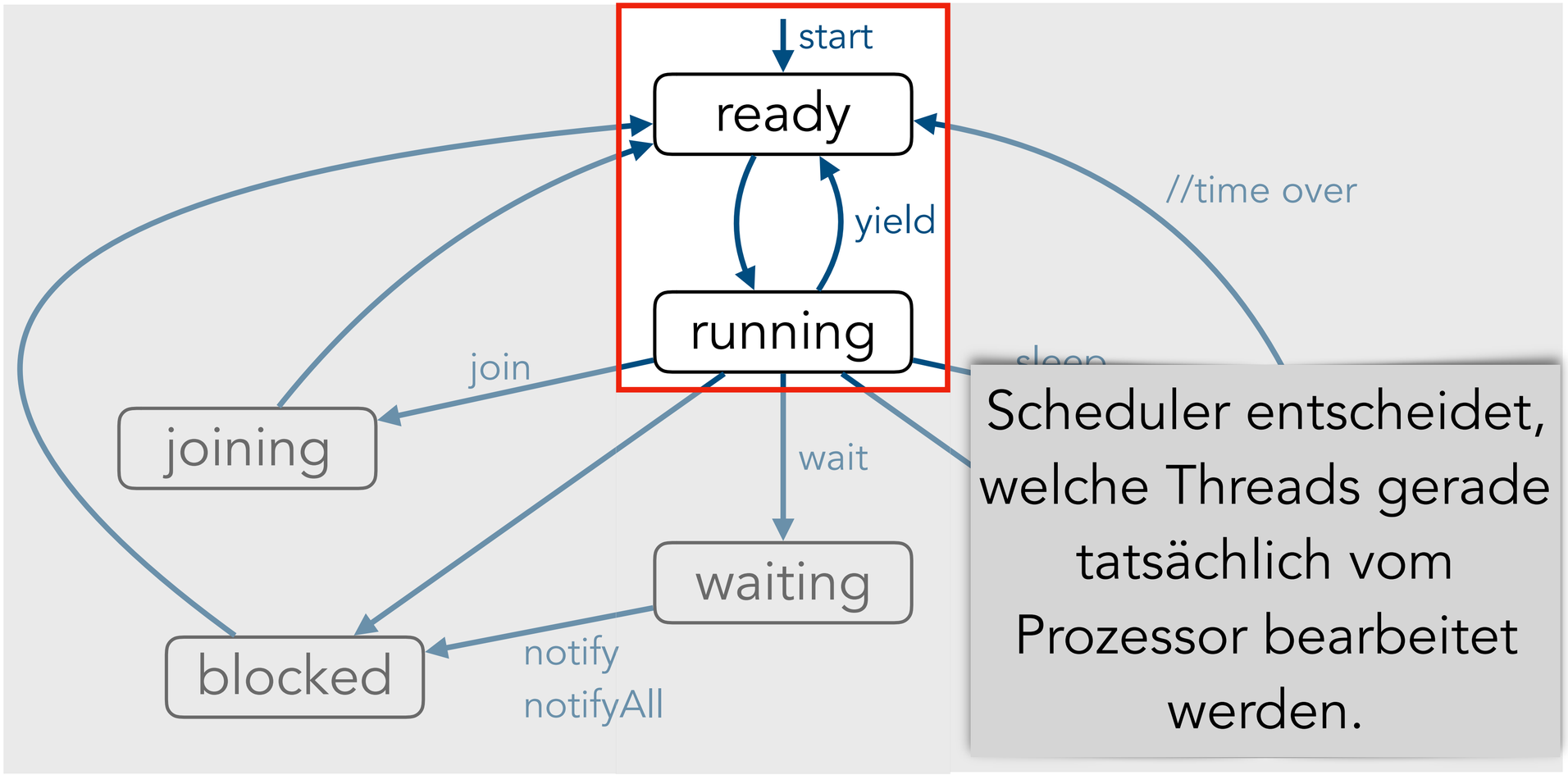
Nebenläufigkeit II

Zustände von Threads:



Nebenläufigkeit II

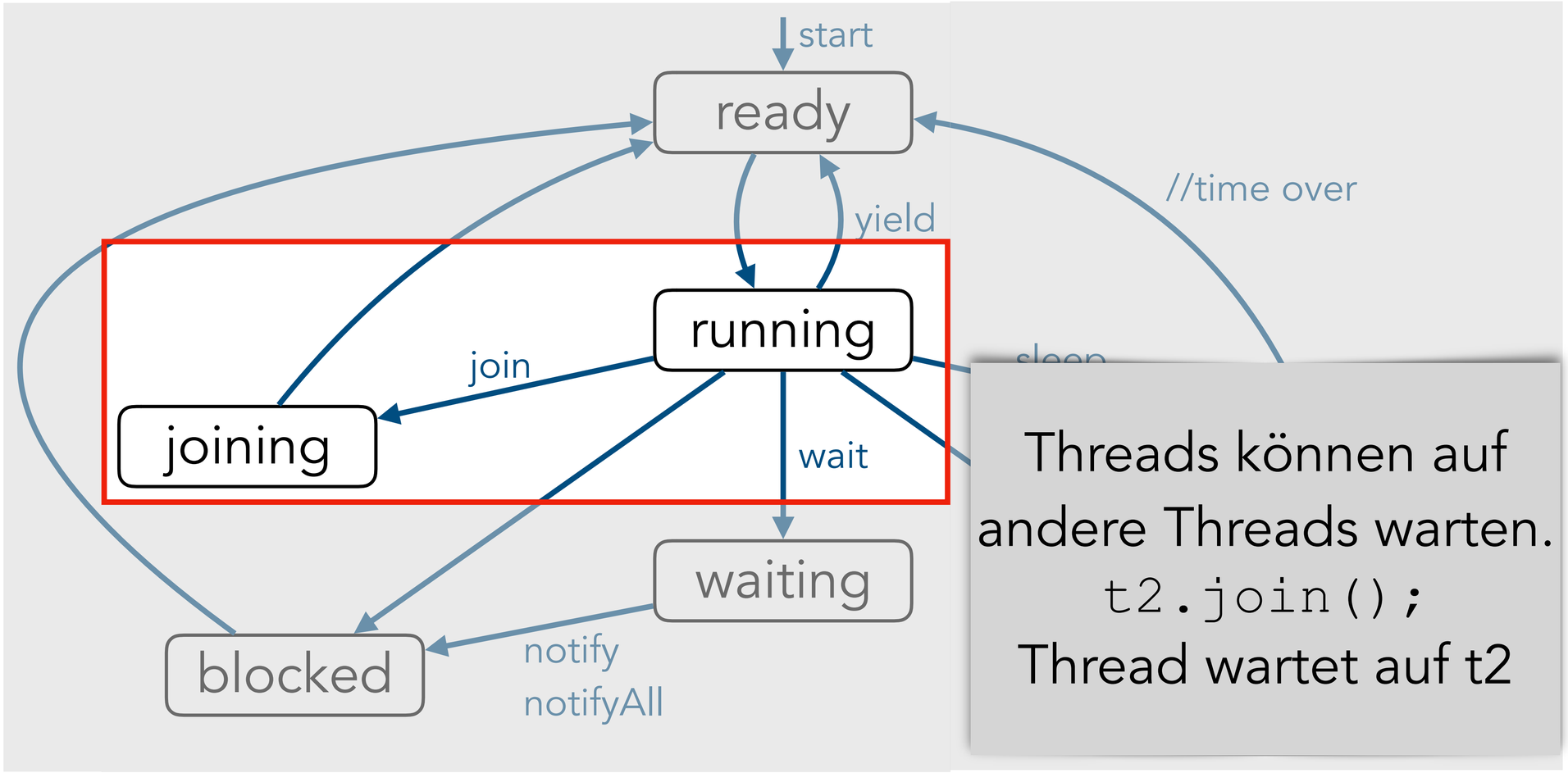
Zustände von Threads:



Scheduler entscheidet, welche Threads gerade tatsächlich vom Prozessor bearbeitet werden.

Nebenläufigkeit II

Zustände von Threads:



Threads können auf andere Threads warten.

```
t2.join();
```

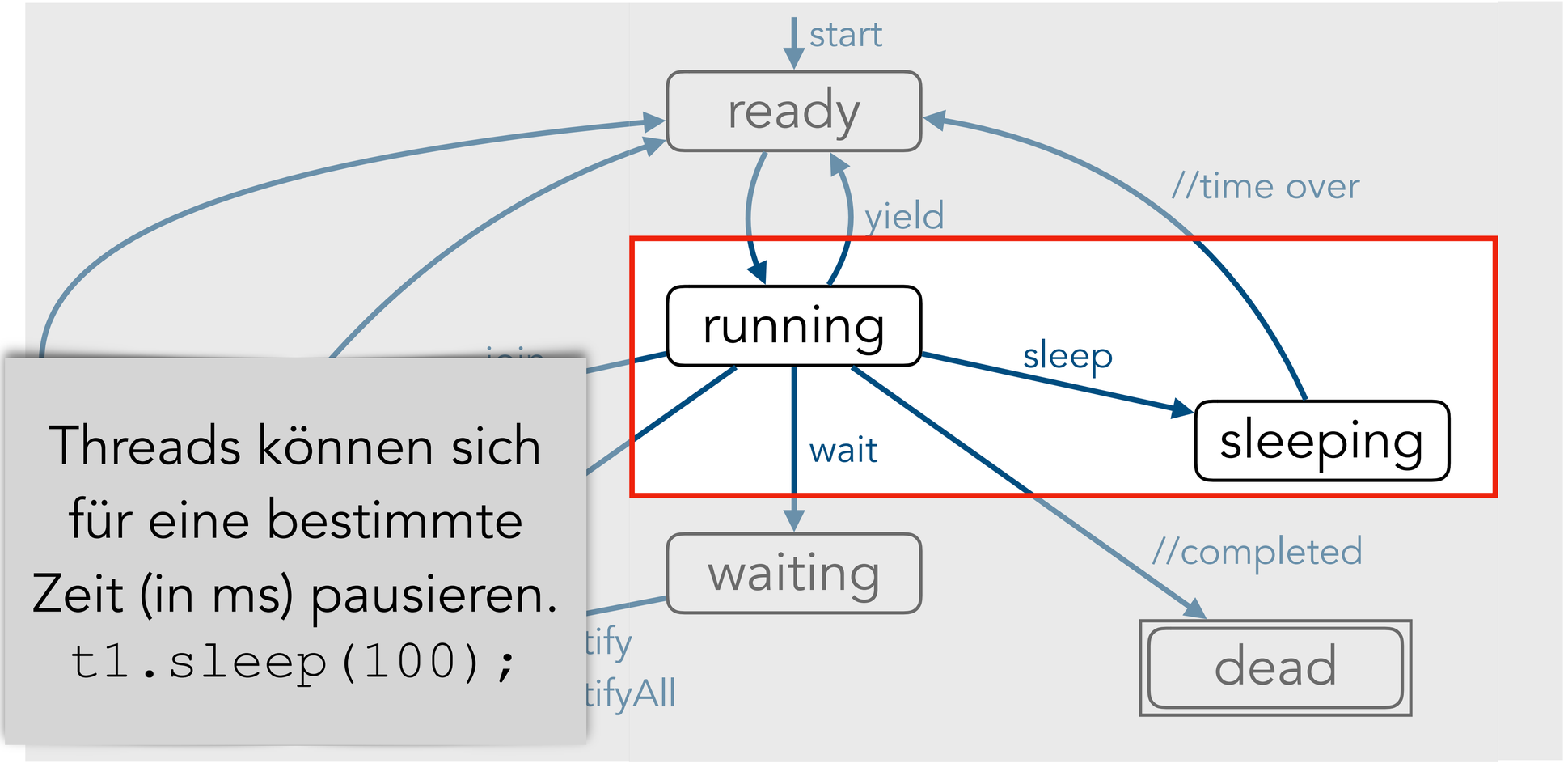
Thread wartet auf t2

Nebenläufigkeit II

Zustände von Threads:

Nebenläufigkeit II

Zustände von Threads:

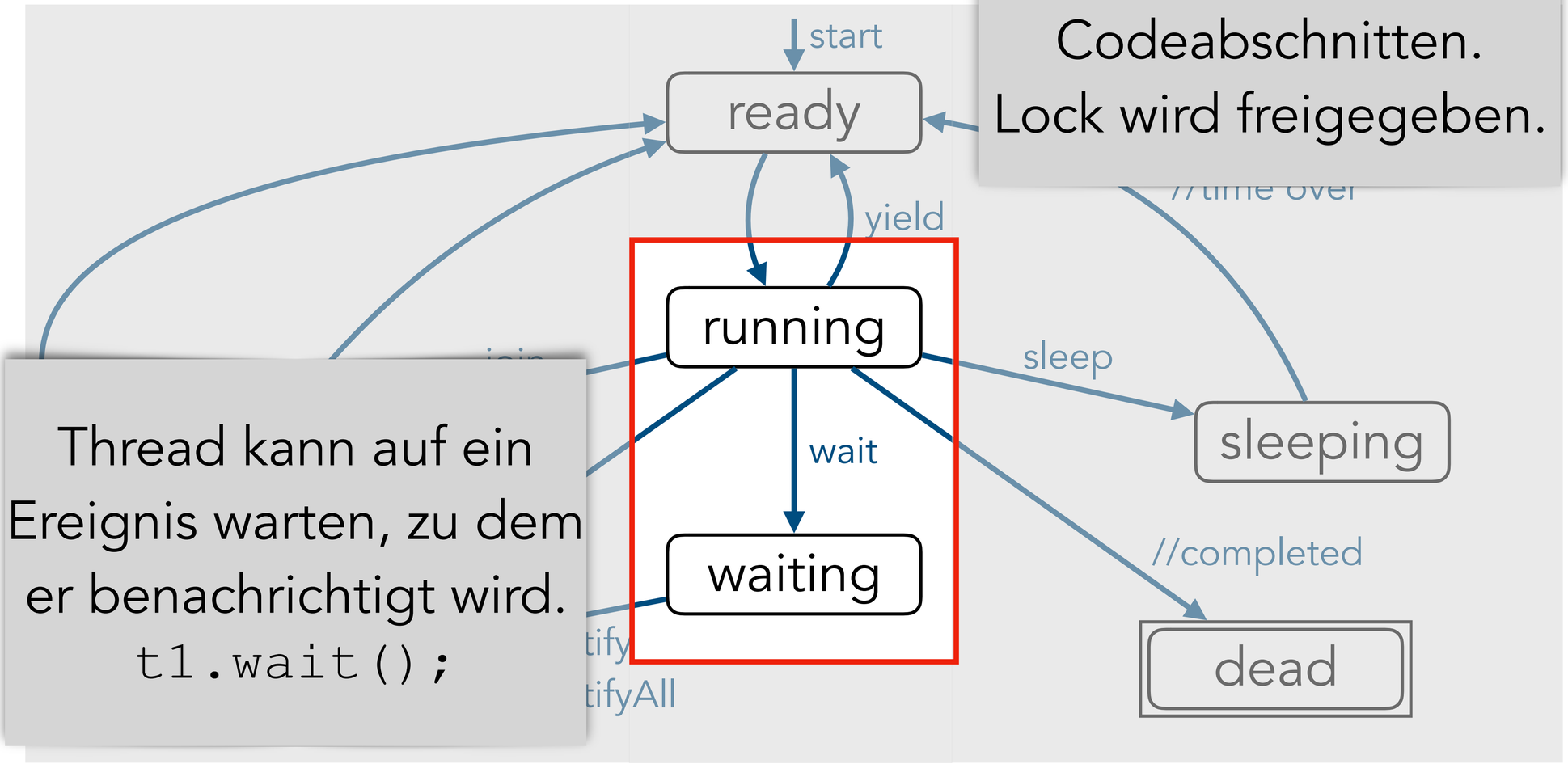


Threads können sich für eine bestimmte Zeit (in ms) pausieren.

```
t1.sleep(100);
```

Nebenläufigkeit II

Zustände von Threads:



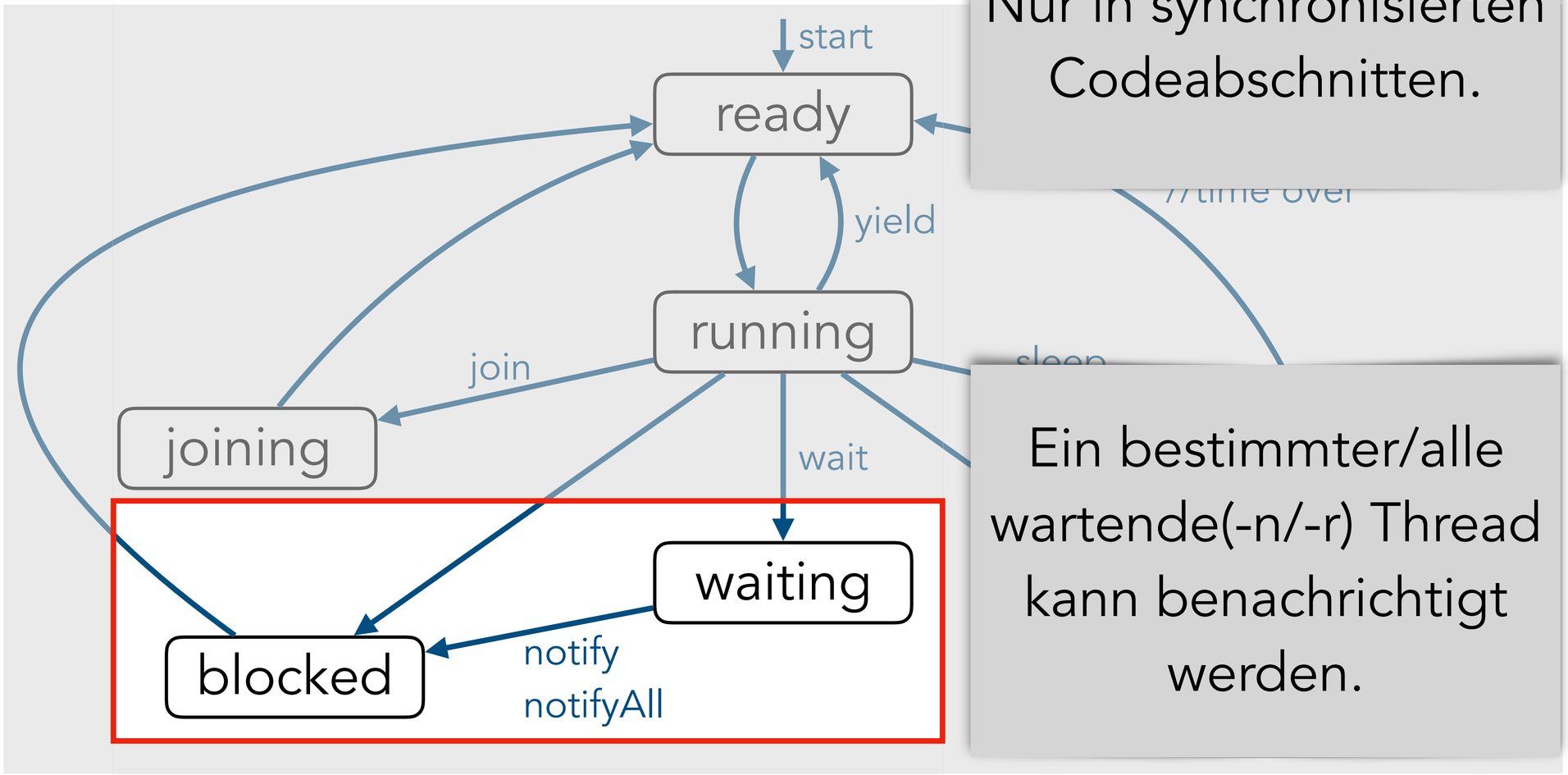
Nur in synchronisierten Codeabschnitten. Lock wird freigegeben.

Thread kann auf ein Ereignis warten, zu dem er benachrichtigt wird.

```
t1.wait();
```

Nebenläufigkeit II

Zustände von Threads:

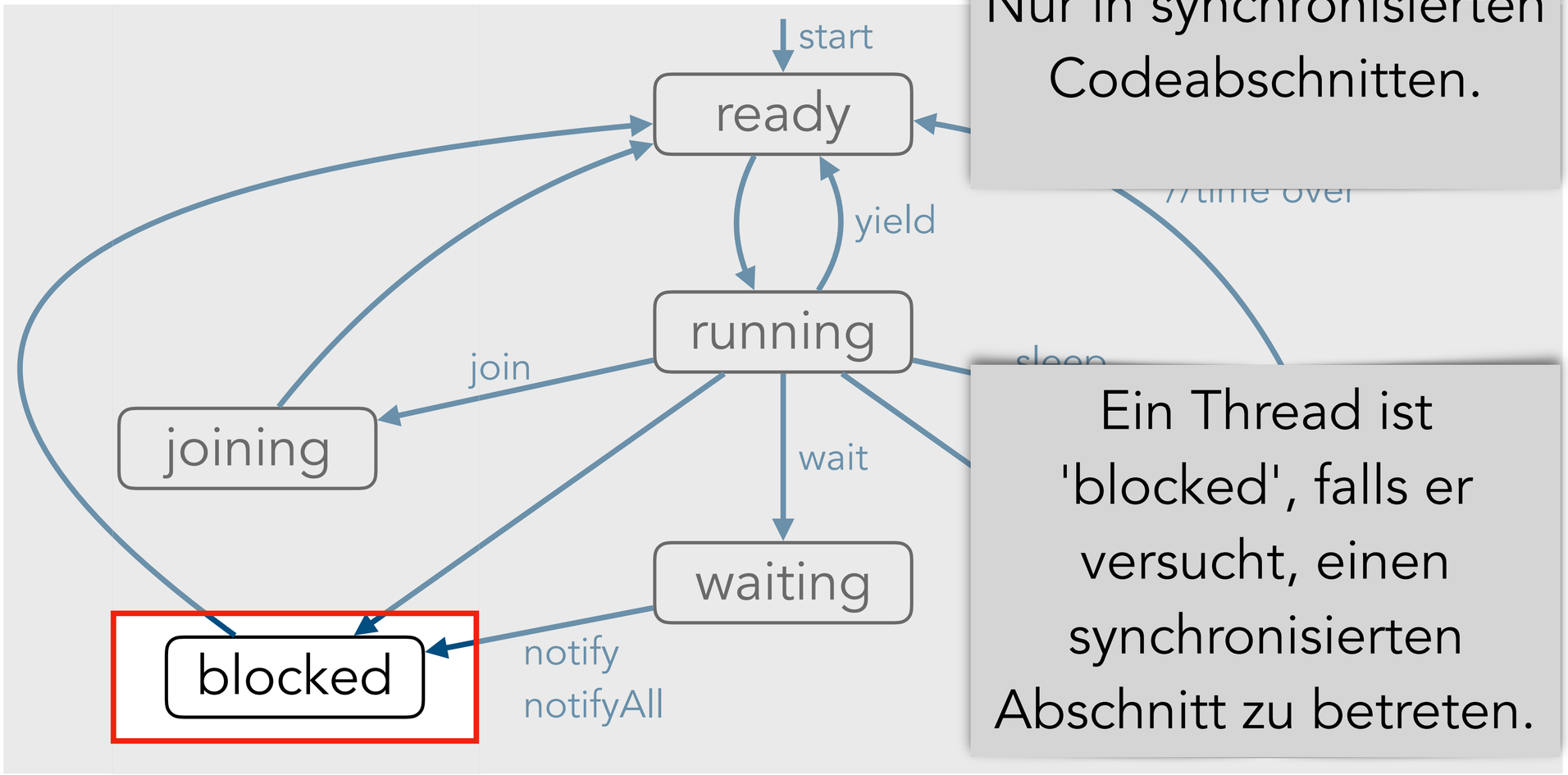


Nur in synchronisierten Codeabschnitten.

Ein bestimmter/alle wartende(-n/-r) Thread kann benachrichtigt werden.

Nebenläufigkeit II

Zustände von Threads:

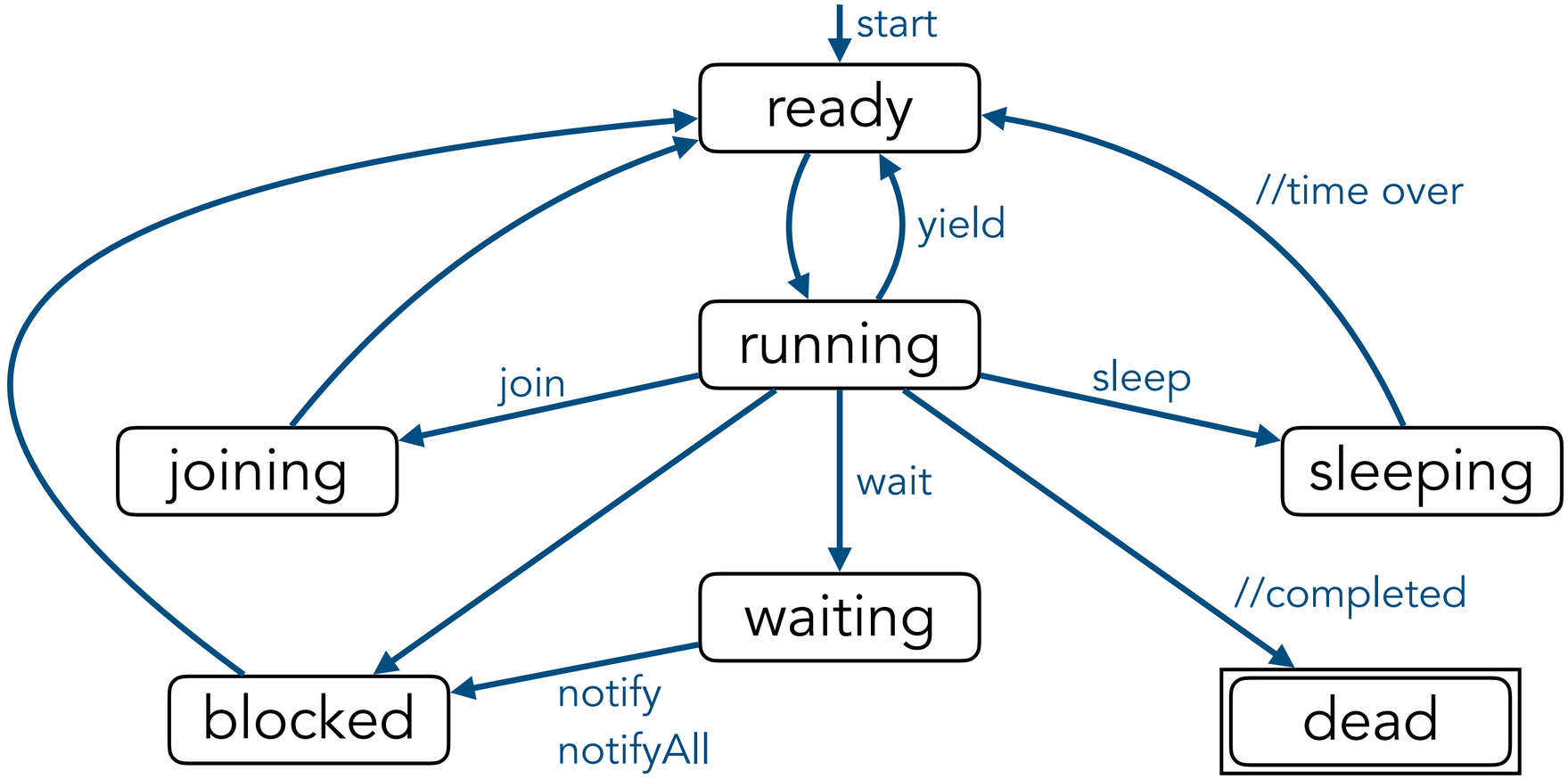


Nur in synchronisierten Codeabschnitten.

Ein Thread ist 'blocked', falls er versucht, einen synchronisierten Abschnitt zu betreten.

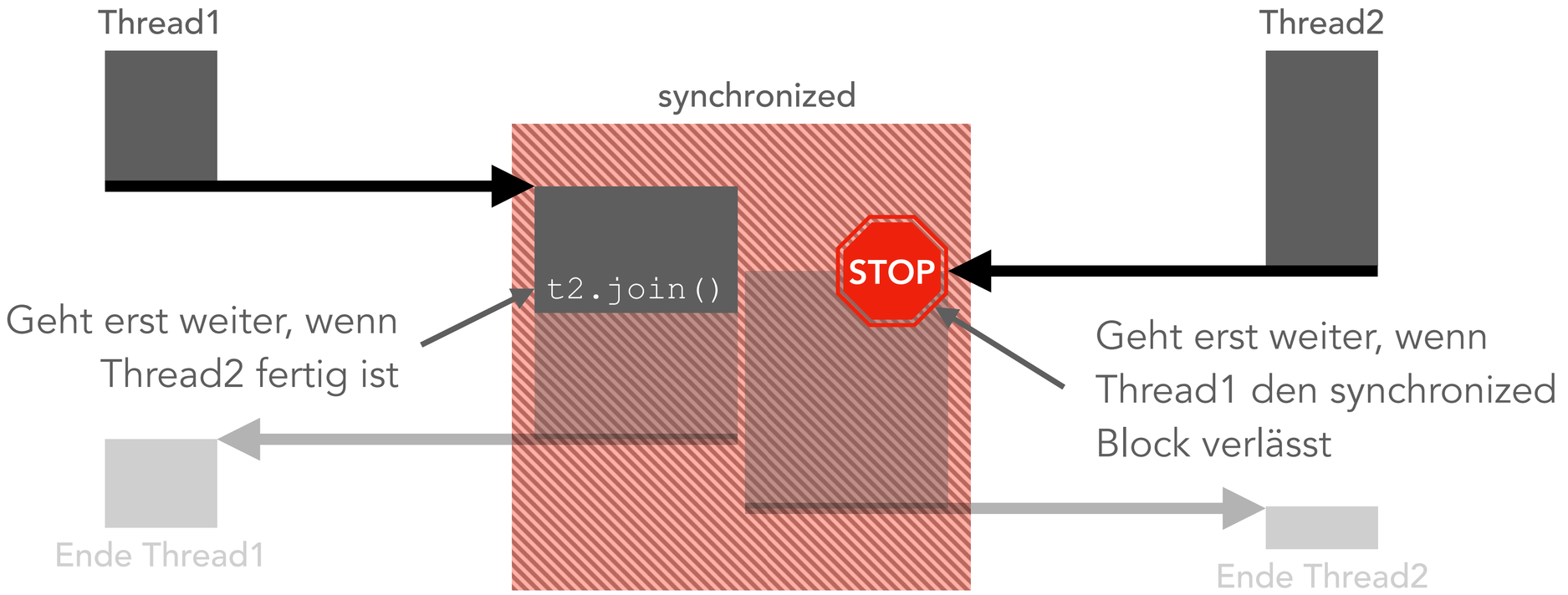
Nebenläufigkeit II

Zustände von Threads:



Nebenläufigkeit II

Vorsicht vor Deadlocks:



Nebenläufigkeit II

Streams parallelisieren:

```
Stream.of(1, 2, 3, 4, 5, 6)
   .parallel()
   .map(i -> i*i)
   .forEach(System.out::println);
```

Parallelsierung braucht auch Ressourcen.

→ eventuell sogar langsamer!

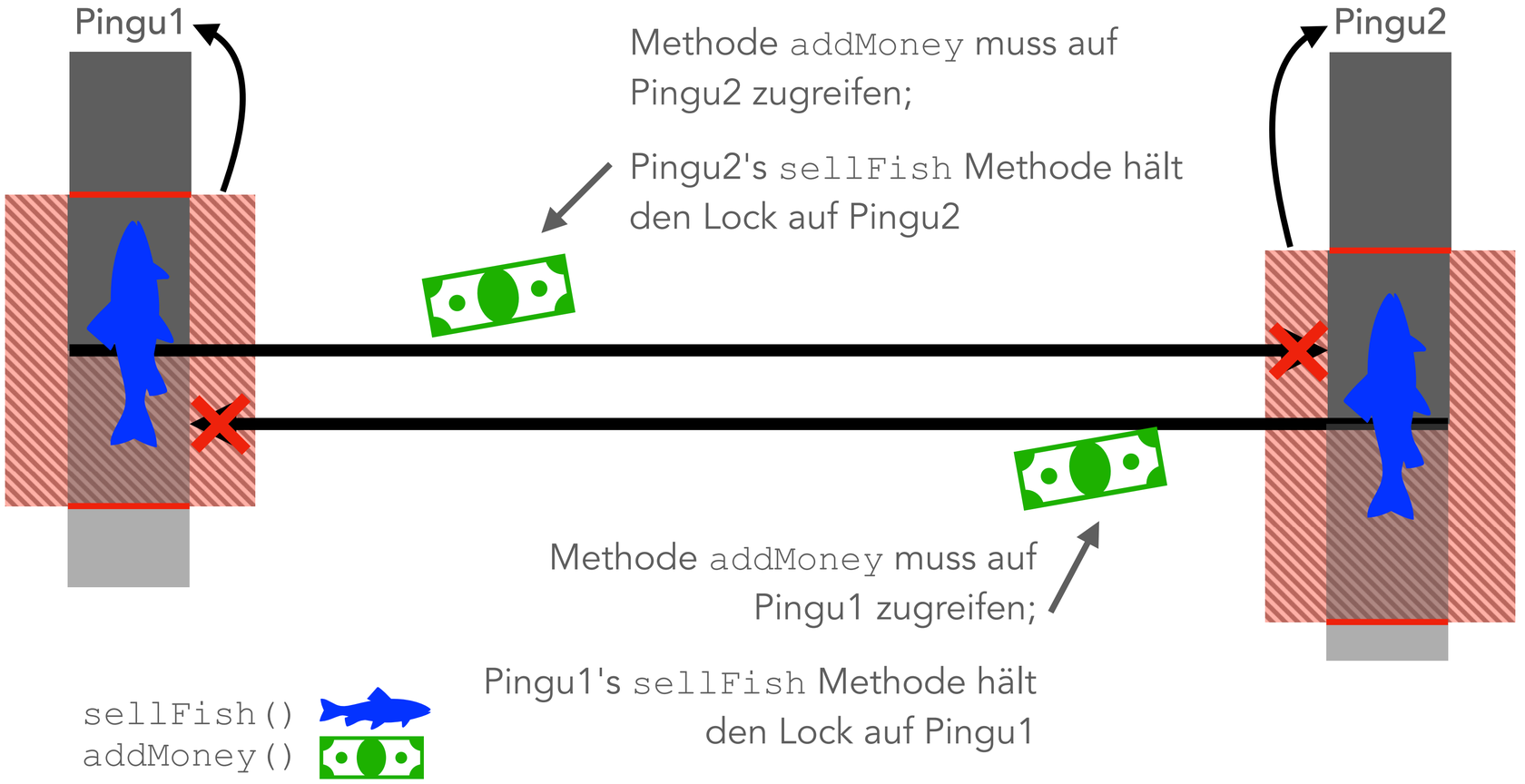
P12.01

Geschäftspartner 2

- Aufgabe von letzter Woche in **`synchronized`**.
- Ist das jetzt richtig?
  > DeadLock?

P12.01

Geschäftspartner 2, Deadlock:



P12.02

Synchrone Sets

- `java.util.LinkedList` 's erlauben keinen synchronisierten Zugriff (sicher vor RaceConditions)
- Ist **`synchronized`** hier Sinnvoll?
  > Nein, gleichzeitiges Lesen sollte möglich sein!
- Implementierung eines RW-Locks
  > Vorsicht vor RaceConditions & Deadlocks!

P12.03

Parallele Streams

- Streams sind bei `.parallel()` sehr intransparent
  > Wir möchten das Verhalten sichtbar machen

```
public static String printlnComparison(final
int numPrintStatements)
```

> jedes mal Name des Threads ausgeben, `Thread.getName()`

```
public static String calcComparison(final
int numCalculations)
```

> parallele Berechnung von Exponentialreihen

$$\sum_{n=0}^{i} e^n$$ mit *Math*.exp

P12.03

Parallele Streams

- Zeit kann gestoppt werden:

```
long start = System.currentTimeMillis();
// Do something
long duration = System.currentTimeMillis() -
  start;
System.out.println(duration);
```

```
public static long countNumThreads(final
int numTasks)
```

> Zahl der Standardmäßig verwendeten Threads

P12.03

Parallele Streams

- Hängt das Verhalten von der Art des ausgeführten Tasks ab?
  > Nein.
- Was folgt daraus? Wann sollte man die parallelisieren von Streams nutzen, wann nicht?
  > Parallelisieren nur für rechenintensive Aufgaben ohne Synchronisierung (wie beim `println()`)

P12.03

Parallele Streams

- Wie viele Threads verwendet `.parallel()` standardmäßig?
  > Anzahl der Kerne des Prozessors (-1, für andere Programme)
- Sind die CPU- bzw. Prozessorkerne auf Ihrem Rechner entsprechend ausgelastet?
  > teilweise, allerdings muss jeder `println()` Befehl synchronisiert werden

P12.04

Klausurkorrektur

- Parallele Korrektur einer Klausur mit verschiedenen Queues und Producer/Consumer.

Siehe Producer/Consumer